MOTOROLA
Semiconductor Products Inc.

Design Concept
DC-001

Virtual Memory Using the MC68000 and the MC68451 MMU

This paper is a Design Concept. It is meant to present useful ideas in design. As such, it represents a thorough discussion of a possible design. However, the system discussed herein has not been built and tested.

Prepared by
Hunter Scales
Systems Applications Engineer
Advanced Microcomputer Components

INTRODUCTION

Early in the history of computers, programmers found that their programs were increasing in size until they were literally larger than the addressing range of the computer. To solve this problem, a technique called virtual memory was invented. This technique allows the programmer to use a larger address space for his programs than the physical address space of the main memory by automatically storing and retrieving parts of the program in secondary memory (usually a disk).

The original virtual memory technique, as implemented by IBM, used a main memory addressing scheme which referenced a page table to get a pointer into a block table. In turn, this pointer was used to form the physical address. This meant that every memory reference required three main memory references. The use of a cache of addresses in an associative memory can cut this time significantly.

This paper presents a design for a virtual memory machine using the currently available MC68000 microprocessor (MPU) and the MC68451 memory management unit (MMU). The presentation includes a discussion on some problems inherent to virtual memory design and the methods used to resolve these problems.

DESIGN GOALS

In this design, the user program (called a task) is allowed to request and receive from the operating system more memory than is physically available. The operating system then allocates some minimum amount of memory to the task and constructs a segment or segments in the MMU to describe the page allocated. If the task then tries to access memory which has not been physically allocated, an undefined segment access error is generated by the MMU. The MMU then asserts the $\overline{\text{FAULT}}$ signal to indicate that a page fault has occurred.

Once a page fault has occurred, some mechanism must be available to locate and fix the fault. This consists of determining if the present page has been modified and, if it has, to save it on the disk. The new page containing the location whose address caused the page fault must then be loaded from the disk into memory. The segment descriptor(s) which describe the page in the MMU must then be modified to reflect the new memory configuration.

Two approaches toward the achievement of these goals are presented. These are the bus cycle rerun method and the bus cycle suspension method.

BUS CYCLE RERUN METHOD

The obvious candidate to fix the page fault is the MPU, as it has access to both the MMU and the DMAC. Unfortunately, the bus cycle which caused the fault must be rerun after the fault has been fixed in order to continue executing the program. The MC68000 can rerun bus cycles by using the bus error ($\overline{\text{BERR}}$) and halt ($\overline{\text{HALT}}$) signals. However, the MPU is in the halt state between the aborted cycle and the rerun cycle and cannot fix a page fault while it is halted. Therefore, another bus master must perform this function and, since this bus master and the main MPU can share memory management routines, this could be an MC68000 MPU as well.

A block diagram of a bus rerun type of system is shown in Figure 1. The MPU labeled Executor serves as the main processor, executing the operating system and the user tasks. The Fixer is responsible for fixing page faults. Since both MPUs share a common bus, the bus request ($\overline{\text{BR}}$) and bus grant ($\overline{\text{BG}}$) control signals are used by the control logic to allow only one MPU at a time to use the bus.

When a page fault occurs, the $\overline{\text{BERR}}$, $\overline{\text{HALT}}$, and $\overline{\text{BR}}$ lines on the Executor are asserted. This causes the current bus cycle to be terminated and the Executor to be halted. As soon as the cycle terminates, the MPU relinquishes the bus. The control logic then releases the Fixer by negating its $\overline{\text{BR}}$ line and the Fixer takes control of the bus and fixes the page fault. After resolving the page fault, the Fixer writes to a special location to toggle a flip-flop which causes the swap to occur. The $\overline{\text{BR}}$ line on the Fixer is asserted and it is removed from the bus. The $\overline{\text{HALT}}$ and $\overline{\text{BR}}$ lines on the Executor are then negated and the Executor performs the bus rerun and then continues executing the user task.

While this method is relatively conservative of hardware, it does have one major drawback. In order to preserve the integrity of semaphores useful in multi-processor applications, read-modify-write bus cycles cannot be rerun. In practice, this means that user applications programs may not use the test and set (TAS) instructions in the bus cycle rerun method. Since it may not be possible to apply this restriction, particularly on vendor-supplied software, another method is proposed.

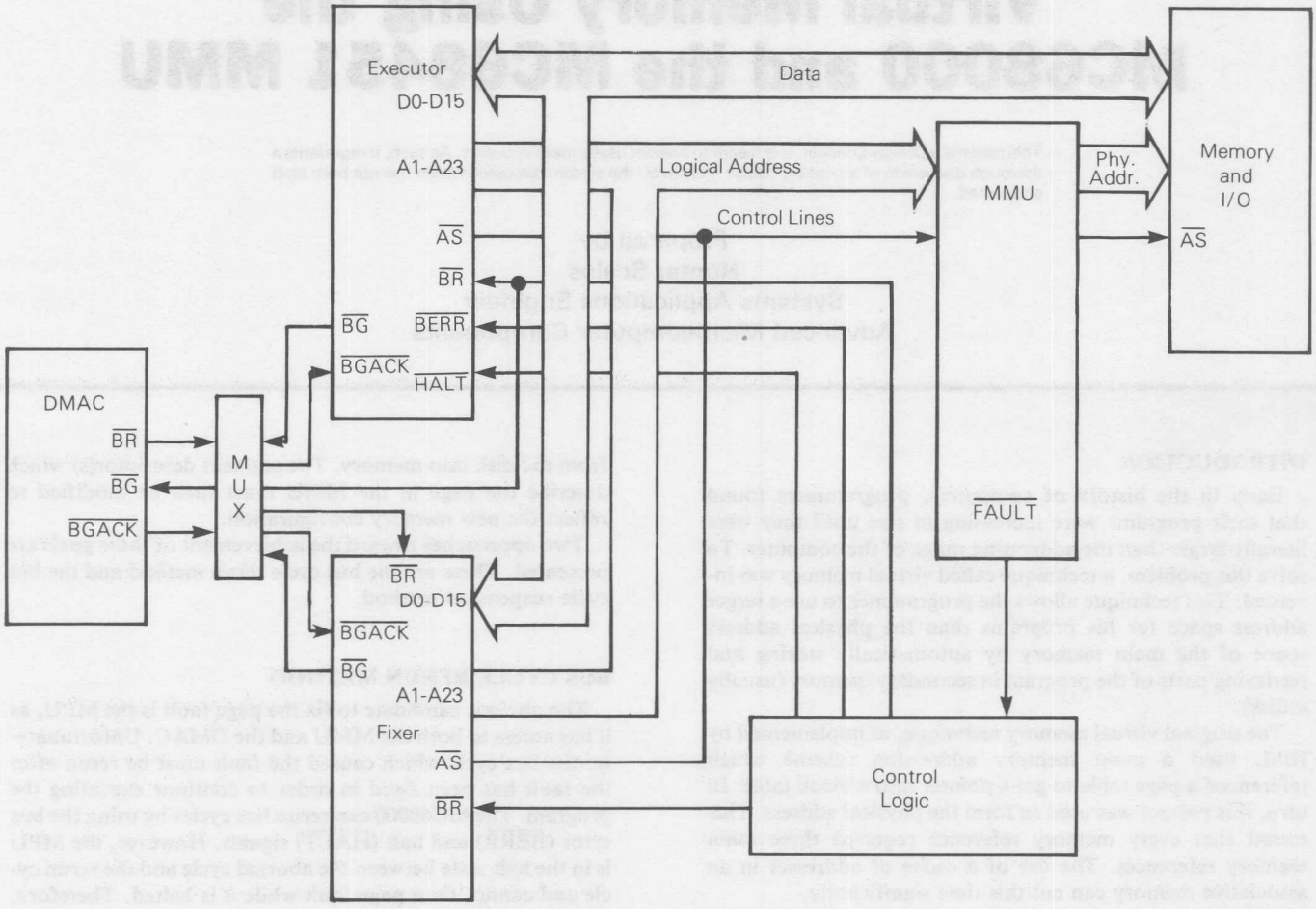


FIGURE 1 — Bus Cycle Rerun System Block Diagram

BUS CYCLE SUSPENSION METHOD

The MC68000 has an asynchronous bus interface. The MPU asserts address strobe ($\overline{AS}$) to indicate a valid address on the address bus and one of the upper or lower data strobes ($\overline{UDS}$) or ($\overline{LDS}$) to indicate valid data during a write cycle. The MPU then expects a data transfer acknowledge ($\overline{DTACK}$) response signal to be asserted, indicating that the data has been accepted (write) or is valid (read). If $\overline{DTACK}$ has not been asserted by the falling edge of state four (S4) of the system clock, the MPU idles, inserting wait states until $\overline{DTACK}$ is asserted. The bus cycle is therefore suspended until $\overline{DTACK}$ is asserted and this delay can be used by the Fixer to fix the page fault.

A block diagram of a system using this method is shown in Figure 2. Since the Executor is driving the address, control and, possibly, the data buses during the "suspension," three-state buffers are needed to isolate these signals from the system bus while the Fixer is active. The Fixer is held off the bus while the Executor is active with the $\overline{BR}$ signal. This signal causes all buses and control signals on the Fixer to enter the high-impedance state and to halt.

When the Executor executes a bus cycle wherein a page fault occurs, the MMU withholds the mapped address strobe ($\overline{MAS}$) and asserts the $\overline{FAULT}$ signal. This action disables the three-state buffers and removes the Executor from the system bus. Since the data transfer acknowledge line on the Executor ($\overline{DTACK}$(E)) is held negated by a pullup resistor, the Executor idles in the wait state. Asserted $\overline{FAULT}$ also negates the $\overline{BR}$ line to the Fixer and releases the Fixer to control the bus. After performing the fix, the Fixer writes to a selected location to cause a swap. Signal $\overline{BR}$(F) is again asserted, removing the Fixer from the bus and allowing the suspended bus access of the Executor to be completed. Signal $\overline{DTACK}$(E) is then asserted by the addressed memory block or peripheral and the cycle wherein the page fault occurred terminates. The Executor then continues with the user task.

The tradeoff in this method is the amount of hardware required versus a versatile instruction handling capability. Although the control logic is less complex, three-state buffers are required for the address, data, and control buses of the Executor. Also, multiplexers are needed for the address and bus request lines. However, all instructions, including TAS, can be used on this system. Thus, this method is preferred as it results in a more powerful and versatile system. A design incorporating this method is described in detail in the following paragraphs.

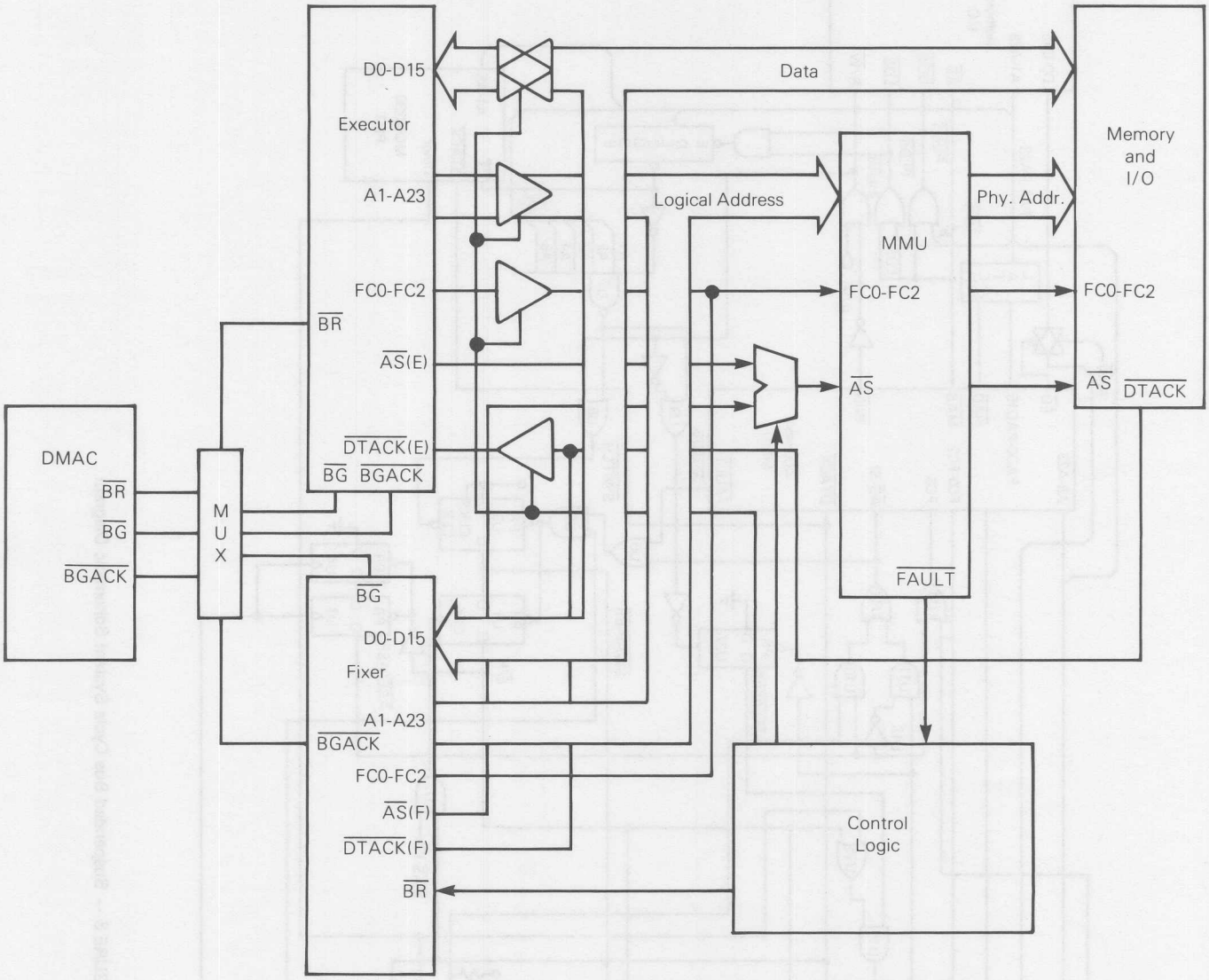


FIGURE 2 — Suspended Bus Cycle System Block Diagram

A DESIGN USING THE CYCLE SUSPENSION METHOD

Figure 3 shows a schematic diagram of a virtual memory machine using two MC68000L8 microprocessors. The Executor is isolated from the system bus by three-state buffers. These buffers are controlled by the $\overline{E}/F$ signal generated by control logic flip-flops U1 and U2. When $\overline{E}/F$ is low, the buffers are enabled and the Executor is in control of the bus. When $\overline{E}/F$ is high, the buffers are in the high-impendance state and the Executor is removed from the bus. The control logic uses the $\overline{SWAP\text{-}BR}$ signal to remove the Fixer from the bus while the Executor is processing.

The system address strobe, $\overline{AS}(S)$, is derived from the multiplexed address strobes of the Executor ($\overline{AS}(E)$) and the Fixer ($\overline{AS}(F)$). The address control logic flip-flops U3 and U4 use the $\overline{ASE}/ASF$ signal to select either $\overline{AS}(E)$ or $\overline{AS}(F)$ as the system address strobe $\overline{AS}(S)$. When $\overline{ASE}/ASF$ is low, $\overline{AS}(S)$ is $\overline{AS}(E)$ and when it is high, $\overline{AS}(S)$ is $\overline{AS}(F)$. This signal is asserted one clock cycle after $\overline{E}/F$ to allow for address setup time to the MMU.

The bus request handshake lines, $\overline{BR}$, $\overline{BG}$, and $\overline{BGACK}$ are similarly multiplexed with gates U18 through U26 to allow the DMAC to request the bus from the current processor in control. The select line of this multiplexer is $\overline{E}/F$.

The MMU can assert $\overline{FAULT}$ for an undefined segment access (USA) and a write violation (WV). This creates a problem in that in this system, a USA may or may not be an error. If the task is attempting to access memory granted to it by the operating system but not physically allocated (virtual memory) then this constitutes a page fault and the Fixer can resolve it. If the access is to a location not requested by the task, the USA is an error and an exception must be forced on the Executor. Similarly, a write violation is an attempt by a user task to write a write-protected segment. This is also an error and must be terminated. The write violation error ($\overline{WVERR}$) signal is used to resolve this problem.

The Fixer can assert the $\overline{WVERR}$ signal by performing a read at a specified location. In turn, $\overline{WVERR}$ asserts the Executor $\overline{BERR}$ line, forcing the Executor to abort the access and take the bus error exception when it regains the bus. Thus, the Fixer is allowed to deal with true USA and WV errors.

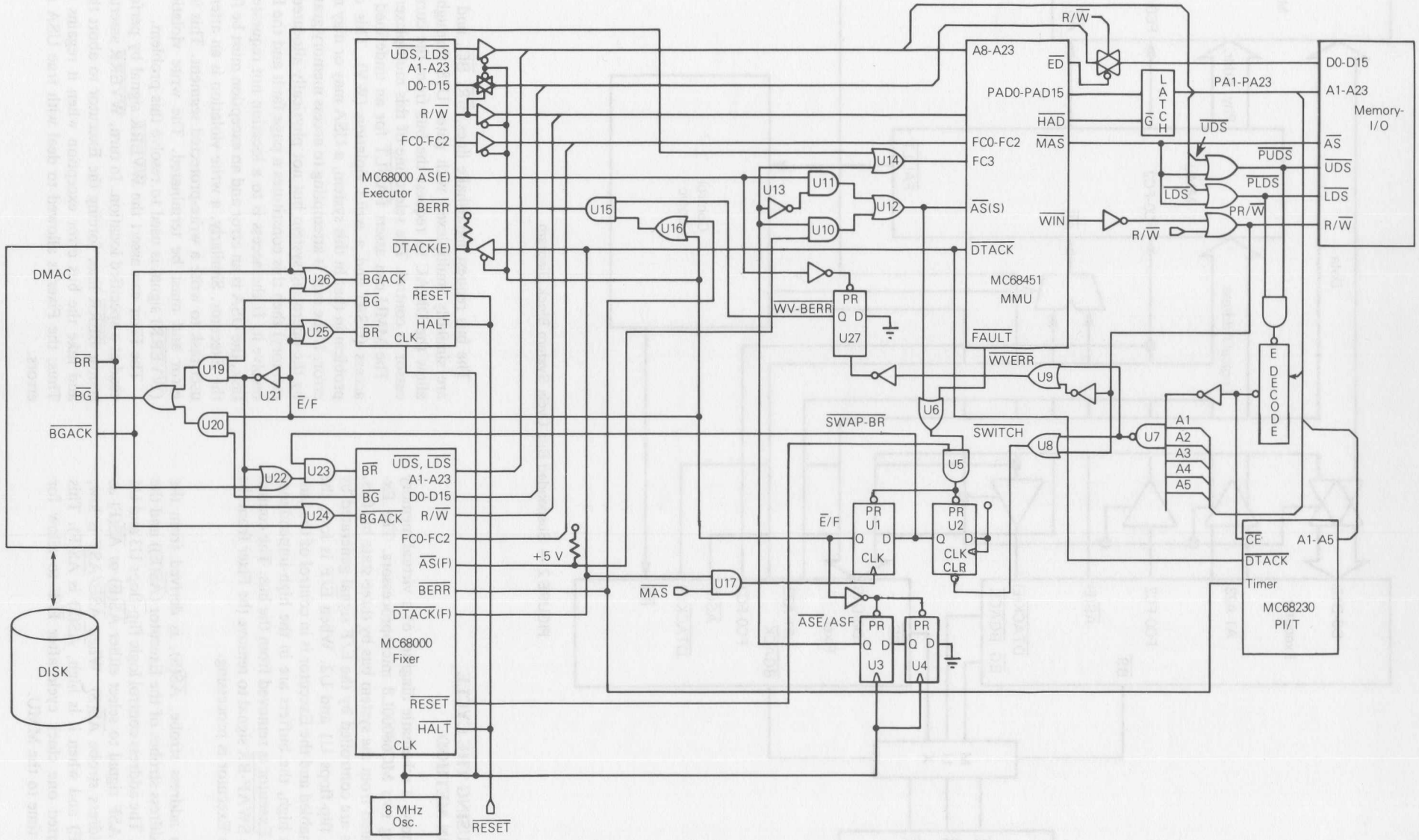
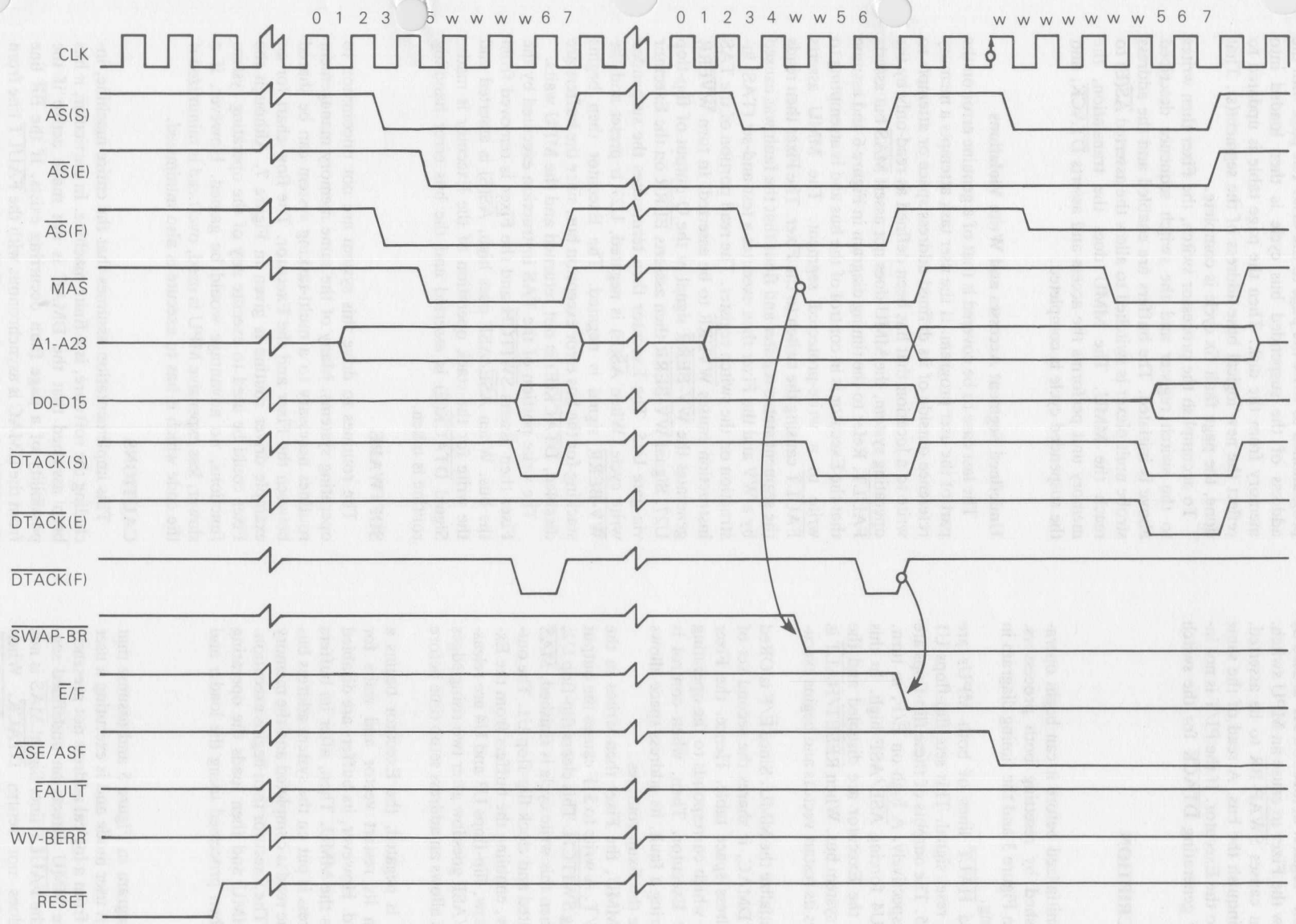


FIGURE 3 — Suspended Bus Cycle System Schematic Diagram



1) Both executor and fixer are in reset, buffer from executor is disabled, $\overline{BR}$ to fixer is negated.
2) Reset is negated. Both MPU attempt to fetch restart vectors.
3) $\overline{DTACK}$ to executor is blocked, vector fetch is suspended.
4) Fixer fetches restart vectors and begins execution.
5) Fixer sets up MMU in preparation for executor vector fetch (different vectors) then branches to fixup routine.

1) Fixer writes to flip register.
2) $\overline{BR}$ to fixer is asserted, fixer relinquishes bus.
3) Buffers from executor are enabled.
4) One clock later $\overline{ASE}$ is asserted to provide $\overline{AS}$ to system.
5) Executor finishes suspended vector fetch.
6) Executor loads operating system and begins operation.

FIGURE 4 — System Reset Timing

An MC68230 Parallel Interface/Timer (PI/T) is provided as a watchdog timer to terminate any accesses to unpopulated addresses. In addition, the PI/T has a number of "null" registers which, when accessed, return DTACK with no other effect. The "null" register at location $1F is used by the control logic to allow the Fixer to cause an MPU switch. A write to this location causes SWAP-BR to be asserted, causing the Fixer to relinquish the bus. A read of the same location asserts BERR to the Executor. If the PI/T is not included, some means of generating DTACK for the switch register must be used.

OPERATIONAL DESCRIPTION

Resetting the System

The system must be initialized before it can begin operation. This is accomplished by resetting both processors. Refer to the schematic in Figure 3 and the timing diagram in Figure 4 for the following.

First the RESET and HALT lines of both MPUs are asserted by the external reset signal. This sets flip-flops U1 and U2 via AND gate U5. The outputs of these flip-flops are SWAP-BR and E/F, respectively. A high on E/F, in turn, sets flip-flops U3 and U4 forcing ASE/ASF high. In this state, the buffers from the Executor are disabled and the Fixer is in control of the system bus. When RESET/HALT is negated, the Fixer fetches its restart vectors and begins execution of the boot ROM.

The fixer must first initialize the MMU. Since E/F is ORed with BGACK from the DMAC, it shares the second set of eight entries in the address space table. Hence, the Fixer should load a descriptor which corresponds to the operating system segment for the Executor. Then, when control is switched to the Fixer during a fault, its address space allows it to address and execute the fixup routines.

After setting up the MMU, the Fixer then writes to the switch location in the PI/T. A write to $1F causes the output of U7 to go low, asserting SWITCH. This clears flip-flop U2, asserting SWAP-BR. When this write cycle is finished, MAS and DTACK(S) are negated and clock flip-flop U1. The output of U1, E/F, goes low, enabling the buffers from the Executor. When E/F goes low, flip-flops U3 and U4 are released from preset and ASE/ASF goes low after two rising edges of the system clock. This allows an address setup time before AS(S) is asserted.

After RESET/HALT is negated, the Executor begins a read bus cycle to fetch its restart vector and waits for DTACK(E) to be asserted. However, its buffers are disabled while the Fixer initializes the MMU. Then, after its buffers are enabled, a valid address is put on the system address bus and AS(S) is asserted. The read is completed and the memory unit asserts DTACK(E). The Executor then begins execution. First, it initializes the MMU and then loads the operating system. User tasks are then processed using the loader and other system utilities.

Page Fault

Refer to the timing diagram in Figure 5 and assume that the Executor is running in user mode and is executing a user task. It attempts to read from a logical address not presently in physical memory. The MMU detects an undefined segment access and asserts the FAULT line. Signal MAS is not asserted, so memory does not return DTACK. When FAULT goes low, U1 and U2 are preset, negating SWAP-BR and forcing E/F high. This removes the executor from the bus and blocks DTACK. The Fixer, with BR released, starts a bus cycle in approximately 3 clock cycles.

The Fixer then reads the MMU to determine what sort of fault caused the switch. If it was a bona fide page fault, it checks the M (modified) bit in the segment status register to see if the segment had been written to. If so, the DMAC is programmed to write the page to the disk. The page with the address of the suspended bus cycle is then loaded into memory from the disk. Then the page table is updated to reflect the new logical base address of the segment(s). That done, the page fault fix cycle is complete.

To accomplish the processor switch, the Fixer then writes to the switch register and the switch sequence described above is initiated. The buffers are enabled and the address strobe multiplexer is switched to allow the asserted AS(E) to reach the MMU. The MMU does the translation, the memory unit performs the access and asserts DTACK, and the suspended cycle is completed.

Undefined Segment Accesses and Write Violations

The last case to be covered is that of a genuine error on the part of the user program. If the user task attempts a memory reference outside of its defined address space or attempts to write to a location that has been defined as read-only by the operating system, the MMU does not assert MAS but asserts FAULT. Refer to the timing diagram in Figure 6 and assume that the Executor is in control of the bus and is attempting to write to a write-protected segment. The MMU asserts FAULT causing the switch to the Fixer. The Fixer then reads the segment status register and finds that the fault was caused by a WV and the Fixer then executes a test-and-set (TAS) instruction on the switch register. The read portion of the TAS instruction causes WVERR to be asserted. In turn WVERR generates the WV-BERR signal at the Q output of flip-flop U27. Signal WV-BERR then assserts BERR on the Executor via gate U15. The Executor then terminates the suspended write cycle. When AS(E) is negated, U27 is preset and the WV-BERR signal is negated. The Executor then begins stacking for the bus error exception but, since the buffers are disabled, DTACK(E) is not returned and the MPU waits.

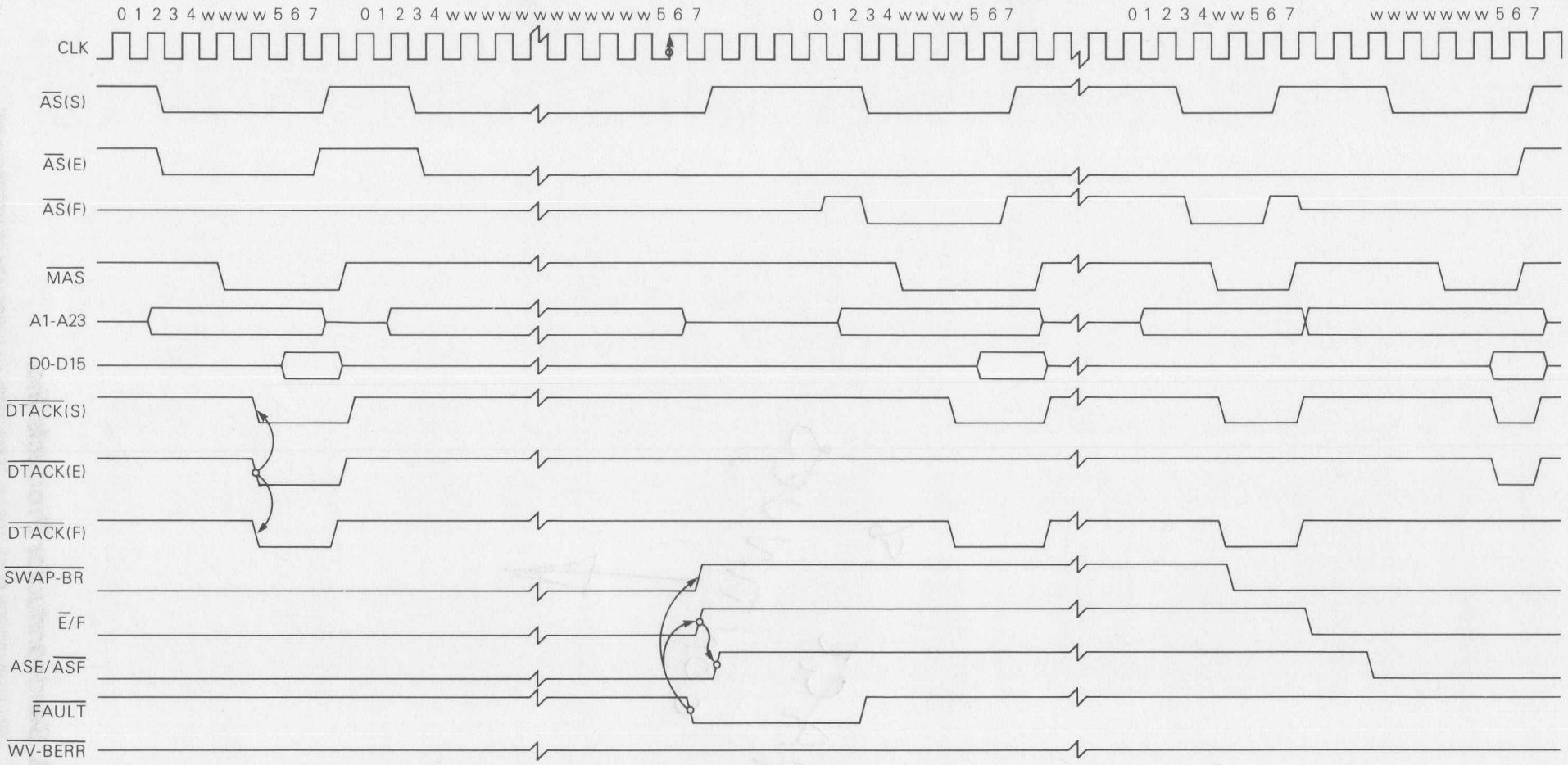
The write portion of the TAS instruction executed by the Fixer then asserts SWITCH and the Fixer is removed from the bus. When ASE/ASF goes high, AS(S) is asserted and the write for the stack operation of the Executor is made. Signal DTACK(E) is asserted and the bus error handling routine is taken.

SOFTWARE

The routines to drive this system are not uncommon to operating systems. Many of the same memory management routines necessary to a multi-tasking system can be shared between the Fixer and the Executor. The flow chart for an example driver routine is given in Figure 7. Although the Fixer could be used to execute any of the operating system functions, no advantage would be gained. However, if a slower, less expensive MPU is used, overhead is minimized if the code which it has to execute is also minimized.

CAUTIONS

This implementation assumes that the entire machine, including the software, is a finite machine. In particular, it has been assumed that the DMAC is not made active if the possibility of a page fault occurring exists. If the BR line from the DMAC is asynchronous with the FAULT line from the MMU, there exists the possibility of the system hanging. This depends on the implementation of the DMAC bus arbitration circuits, as it is possible to supply a BG signal without a BR from the DMAC.



1) Executor in control of bus.
2) Fixer is suspended with BR line.
3) AS of executor is selected AS.

1) Executor reads from undefined segment.
2) MMU asserts FAULT.
3) FAULT releases BR to fixer.
4) Buffers from executor are three-stated, since DTACK(E) is not returned, cycle is suspended.
5) System AS is switched to fixer AS.

1) Within 3 clock cycles fixer starts a bus cycle.
2) Fixer swaps pages to disk and sets up MMU and page table to reflect new configuration.
3) Fixer executes TAS to switch back. E/F is cleared to enable executors buffers.
4) AS(E) is asserted 1 cycle later to allow address setup time.
5) MMU does translation and DTACK is returned to executor, completing suspended bus cycle.

1) Fixer writes to flip register.
2) Fixer BR is asserted and fixer releases control at end of write cycle.

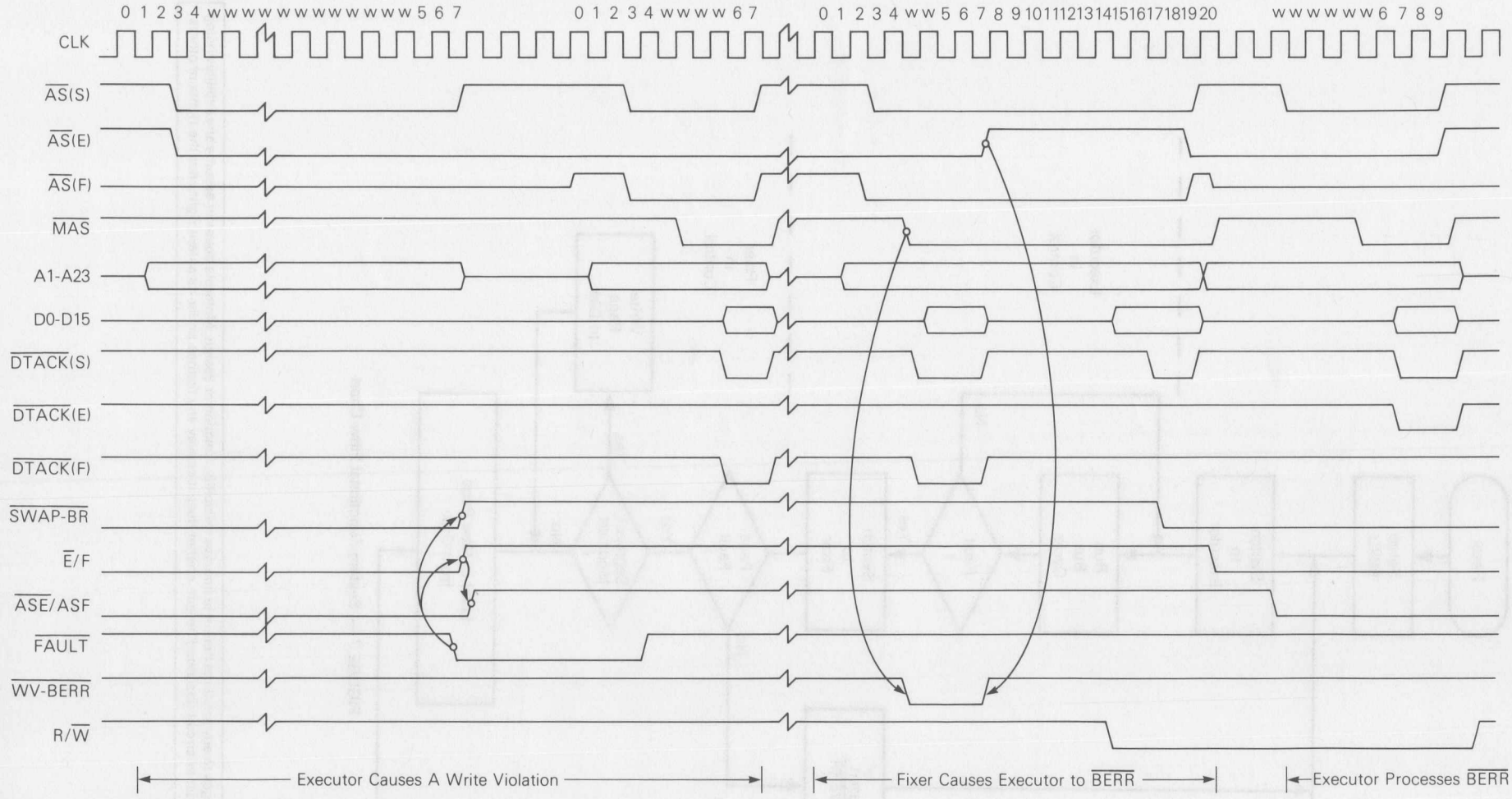
FIGURE 5 — Page Fault Fix Timing

PAGE 8 CONTINUES

MOTOROLA *Semiconductor Products Inc.*

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721 • A SUBSIDIARY OF MOTOROLA INC.

A14924 PRINTED IN USA 3-82 IMPERIAL LITHO C04111 20,000



1) Executor attempts to write to a write-protected location (in error).
2) MMU asserts FAULT while withholding MAS. (DTACK(E) is not asserted).
3) Executor's buffers are disabled; executor bus cycle is suspended.
4) AS(S) is switched to AS(F).
5) BR to fixer is negated (fixer is released).

1) Fixer reads MMU and finds write violation rather than page fault.
2) Fixer executes TAS instruction on flip register.
3) The read portion of the read-modify-write causes BERR on the executor to be asserted.
4) The executor terminates its suspended write cycle and begins to stack for bus error exception processing.
5) The write portion of the TAS instruction causes SWAP-BR to be asserted; fixer releases bus.

1) Executors buffers are enabled and DTACK(E) is returned for BERR stack operation.
2) Executor continues with bus error exception processing to handle the write violation.

FIGURE 6 — Write Violation and Bus Error Timing

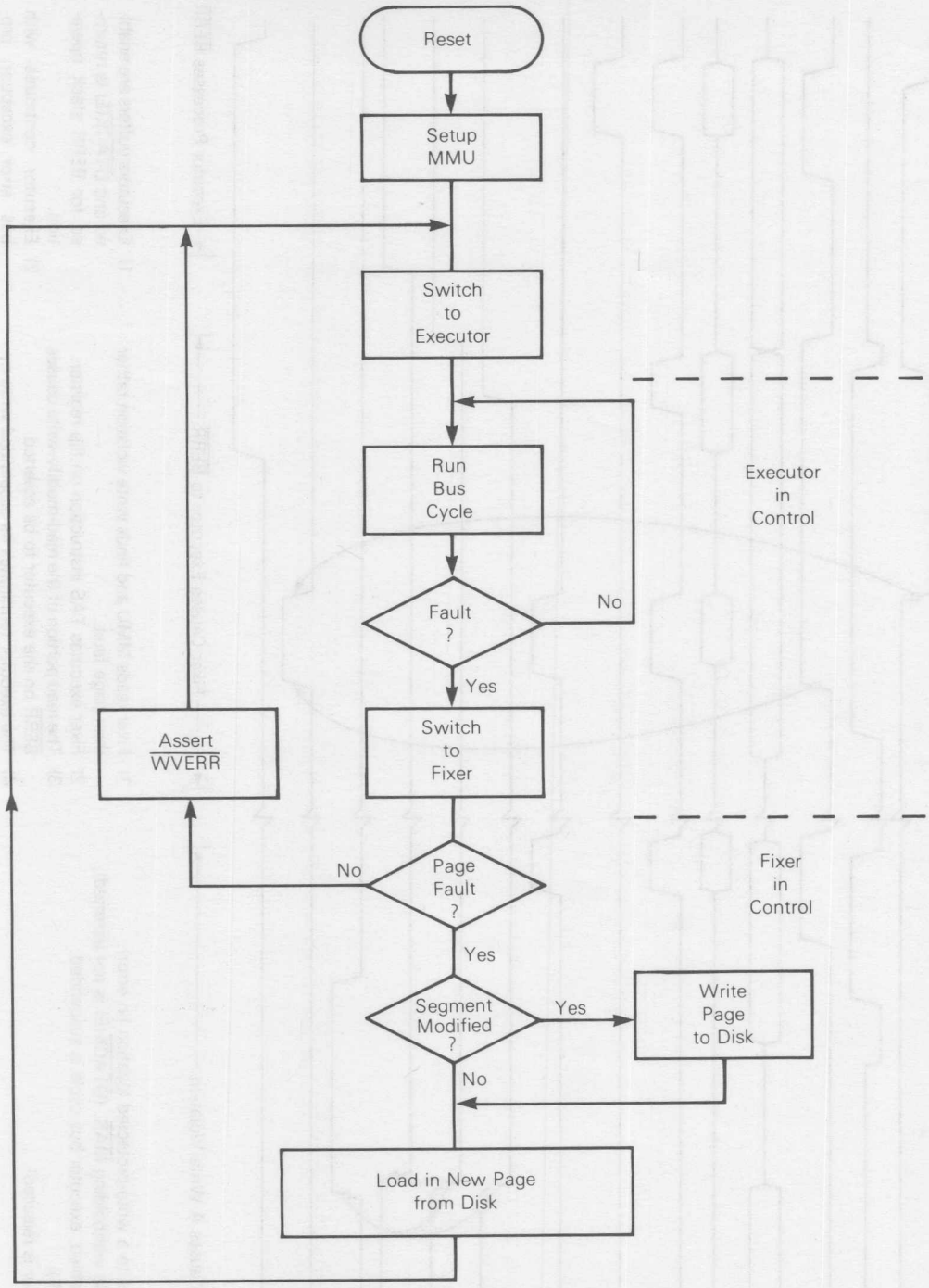


FIGURE 7 — System Functional Flow Chart